OPTIMIZING PANDAS CODE FOR HETEROGENEOUS BACKENDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Data scientists rely on pandas for building data-processing workflows due to its expressive API and rich ecosystem, but Python's Global Interpreter Lock and single-threaded execution model make pandas a performance bottleneck at scale. Although modern accelerators like GPUs offer massive parallelism, naively offloading pandas workloads often fails to achieve speedups. These failures are primarily caused by inefficient idiomatic code and greedy scheduling strategies that overlook data-transfer overheads. We propose PANDAX, a system for automatic optimization and heterogeneous scheduling of pandas workflows across CPU and GPU backends. PANDAX uses a multi-agent LLM-based rewriting framework to transform inefficient pandas code into backend-targeted, high-performance equivalents, guided by profiling feedback, correctness validation, and formal equivalence checking. It then applies a learned cost model and a dynamic programming scheduler to assign each code block to the backend that minimizes end-to-end execution time, explicitly accounting for computation and data-transfer costs. To make optimization practical, PANDAX performs rewriting and scheduling using small data samples and extrapolates performance to full datasets via learned execution and transfer models. Experimental results on 25 real-world data-science workflows and 22 TPC-H queries show that PANDAX achieves up to 7.6× speedup over pandas on real workloads and 10.5× on TPC-H, significantly outperforming existing GPU-accelerated pandas frameworks and bridging the gap between ease of use and high-performance heterogeneous execution.

1 INTRODUCTION

Data science workflows are increasingly central to decision-making across domains such as finance, healthcare, retail, and technology. These workflows are commonly implemented using pandas, an open-source Python library that provides high-performance, easy-to-use data structures and tools for data analysis. **As the most widely used data science library, with over 674 million downloads in the last month alone pys and used by 77% of data scientists jet, its performance is a critical, use-inspired problem that directly impacts real-world productivity.** At its core lies the DataFrame (DF), a two-dimensional, tabular data structure with labeled axes for rows and columns. Its popularity stems from 1) an intuitive DF API, 2) seamless interoperability with other Python libraries such as NumPy, scikit-learn, and Matplotlib, and 3) extensive operator support.

However, pandas is fundamentally constrained by Python's Global Interpreter Lock and single-threaded execution model, making it a performance bottleneck for large-scale datasets. To address these limitations, a number of alternative pandas implementations have emerged, such as Modin Petersohn et al. (2020), Dask dask (2015), cuDF cuDF (2018), hipDF hipDF (2025), and Polars polars (2020). These implementations work by utilizing multiple CPU cores or by leveraging heterogeneous backends to improve performance, for instance offloading code to accelerators such as GPUs. While such implementations offer impressive speedups, they suffer from two key issues.

First, idiomatic pandas code is not accelerator-friendly: patterns such as row-wise loops, `apply` with Python UDFs, and imperative control flow prevent vectorization and GPU execution, causing frequent CPU fallbacks and expensive host–device transfers. Second, existing systems adopt greedy scheduling, offloading operations to accelerators even when CPU execution is more efficient due to kernel overheads, unsupported operators, and data-movement costs. Many production workflows

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

are *legacy pipelines* written for CPU-only execution; they predate GPU availability and were never designed for heterogeneous backends, making migration challenging.

In this paper, we introduce PANDAX, which uses a two-phase approach to optimize data science workflows for heterogeneous execution. In the first phase, PANDAX uses an agentic, LLM-based system to rewrite pandas programs to find functionally equivalent programs that can exploit different backends. Prior work has explored pattern-matching-based approaches Baziotis et al. (2024) for rewriting inefficient pandas programs but these approaches do not scale. As these rules are backend-specific, they often break as they fail to capture the diverse syntactic variants of the same program (e.g., loops can be expressed using `for` or `while`). They also require extensive domain expertise to craft manually. PANDAX instead leverages LLMs for this, as they have shown tremendous progress in code generation. However, due to limitations in context length, directly prompting an LLM with a large workflow to optimize is intractable. PANDAX hence uses a novel multi-agent approach to generate CPU- and GPU-specific rewrites for each code block, validates each rewrite, and repairs it as needed.

At the end of the process, the rewriter generates optimized code blocks for both CPU and GPU. The rewrites are also *formally verified* for functional equivalence using a bounded program verifier. The scheduler then determines where to run each block, and reassembles the code blocks into an optimized workflow. To do so, we use a learned cost model to estimate the per-block execution times on CPU and GPU, as well as the cost of transferring data between CPU and GPU. Using a dynamic programming algorithm (DP), we compute for each block the placement that minimizes the sum of its compute plus the data-transfer costs. This results in a heterogeneously scheduled plan that further optimizes the end-to-end workflow.

We evaluate PANDAX on 25 real-world data science pandas workflows and the 22 standard TPC-H queries, achieving speedups of **7.6**$\times$ and **10.5**$\times$ over pandas, respectively.

In summary, we make the following contributions in this paper:

- We formulate pandas workflow optimization as an *agentic rewriting problem*, using a multi-agent LLM system to generate performant, reliable backend-targeted code. To the best of our knowledge, we are the first to enable **verified pandas optimization** by formalizing pandas semantics in SMT, allowing for bounded equivalence checking of code rewrites.
- We propose learned cost models to determine code scheduling on heterogeneous backends, decomposing cost into computation and data-transfer costs, and formulating the scheduling problem using dynamic programming.
- The key novelty of our system, PANDAX, lies in unifying semantic code rewriting, heterogeneous scheduling, and low-overhead formal verification into a single automated pipeline. Evaluations on real-world workloads demonstrate substantial speedups, achieving up to **37**$\times$ over pandas and **27.7**$\times$ over prior systems on heterogeneous CPU/GPU backends.

2 OVERVIEW

In this section, we present an end-to-end example illustrating how PANDAX accelerates data science workflows on heterogeneous backends.

Consider the three code blocks extracted from a Kaggle notebook in Fig. 1a. Code block 1 computes the average `val` for a set of keys by iterating over each key, filtering the DataFrame, and aggregating the results into a new table `res`. Code block 2 performs data cleaning by sorting `res` and applying a string transformation that truncates long keys. Finally, code block 3 visualizes the resulting distribution using a histogram over average `val`.

Executing this workflow on GPU using NVIDIA's cuDF exposes two common inefficiencies. In code block 1, each loop iteration triggers multiple GPU kernel launches, so fixed launch overhead dominates performance (Fig. 1d). In code block 2, cuDF executes `sort_values` on the GPU but falls back to the CPU for `apply`, since arbitrary Python UDFs cannot run on the GPU. These intra-block fallbacks induce GPU$\leftrightarrow$CPU transfers, negating parallelism benefits.

PANDAX automatically optimizes such workflows through a combination of code rewriting and scheduling. It first analyzes each code block and profiles execution on small CPU and GPU samples

```python
# code block 1
metrics = []
for k in keys:
    df_k = df[df["k"] == k]
    metrics.append(sum(df_k.val) / len(df_k))
res = pd.DataFrame({"k": keys, "m": metrics})

# code block 2
res = res.sort_values("k")
cut = lambda s: s[:4] if len(s) > 10 else s
res["k"] = res["k"].apply(cut)

# code block 3
plt.hist(res["m"], bins=10)
```

(a) Original pandas workflow.

```python
# code block 1
res = df.groupby("k").val.mean().reset_index(name="m")

# data movement to CPU
res = res.to_pandas()

# code block 2
res = res.sort_values("k")
cut = lambda s: s[:4] if len(s) > 10 else s
res["k"] = res["k"].apply(cut)

# code block 3
plt.hist(res["m"], bins=10)
```

(b) Rewritten code with scheduling.

Block	CPU		GPU	
	orig	rew.	orig	rew.
1	5010	264	1810	15
2	7920	7920	14400	14400

(c) Execution timings in ms.

CB1 (for)		CB2 (apply)		CB3 (groupby)	
Func.	GPU	Func.	GPU	Func.	GPU
getattr	19	sort_values	1	groupby	1
getitem	36	apply	0	mean	1
eq	18	getitem	0	astype	1

(d) GPU calls per function.

Figure 1: Unoptimized (a) vs. PANDAX-optimized (b) pandas code. Timings in (c) show speedup; profiling in (d) shows fewer GPU calls.

to collect performance signals. It then invokes a *multi-agent LLM* to rewrite each code block into backend-targeted implementations. For code block 1, PANDAX replaces the loop with a vectorized `groupby().mean()` form (Fig. 1b), which reduces GPU calls and improves parallelism (Fig. 1c).

After rewriting, PANDAX chooses where each code block should run to minimize end-to-end runtime, rather than greedily offloading everything to the GPU. This decision must account for both per-block execution time and GPU↔CPU transfer costs for live data. Because PANDAX profiles only small samples, it uses two learned cost models: (i) an online extrapolation model that fits runtime vs. input size from multiple sub-samples to predict full-data CPU/GPU times, and (ii) an offline-calibrated transfer model that predicts GPU↔CPU movement time from DataFrame size and types. Using these estimates, PANDAX solves for the lowest-cost heterogeneous plan via dynamic programming. In this example, code block 1 is cheapest on GPU after rewriting, while code block 2 is cheaper on CPU despite GPU support due to fallback behavior and transfer overheads. As a result, the DP scheduler assigns code block 1 to the GPU and code block 2 to the CPU, yielding a net **1.5× end-to-end speedup** for the workflow despite heterogeneous execution.

3 PROBLEM FORMULATION

We model a data science workflow as n sequential code blocks $b_1, \ldots, b_n$ executing on a heterogeneous backend with one CPU and one GPU. Let the decision variable $x_i \in \{0, 1\}$ indicate the backend assignment for block b_i ($0 = \text{CPU}, 1 = \text{GPU}$). Each block b_i has execution times r_i^{C} and r_i^{G}, while switching devices between consecutive blocks incurs data transfer costs $t_{i-1,i}^{\text{C}\to\text{G}}$ and $t_{i-1,i}^{\text{G}\to\text{C}}$.

To calculate the total runtime, each block i has a start time s_i and completion time C_i, defined recursively as $C_i = s_i + (1 - x_i)r_i^{\text{C}} + x_i r_i^{\text{G}}$, with base case $s_1 = 0$. For subsequent blocks ($i > 1$), the start time includes the previous block's completion time plus any device-switching penalty: $s_i = C_{i-1} + \text{transfer}(x_{i-1}, x_i)$, where $\text{transfer}(x_{i-1}, x_i) = (1 - x_{i-1})x_i\, t_{i-1,i}^{\text{C}\to\text{G}} + x_{i-1}(1 - x_i)\, t_{i-1,i}^{\text{G}\to\text{C}}$.

Our objective is to find the globally optimal placement sequence $x_1, \ldots, x_n$ that minimizes the total end-to-end workflow execution time C_n:

$$\min_{x_1,\ldots,x_n} \sum_{i=1}^{n} \left[(1 - x_i)r_i^{\text{C}} + x_i r_i^{\text{G}}\right] + \sum_{i=2}^{n} \left[(1 - x_{i-1})x_i\, t_{i-1,i}^{\text{C}\to\text{G}} + x_{i-1}(1 - x_i)\, t_{i-1,i}^{\text{G}\to\text{C}}\right] \tag{1}$$

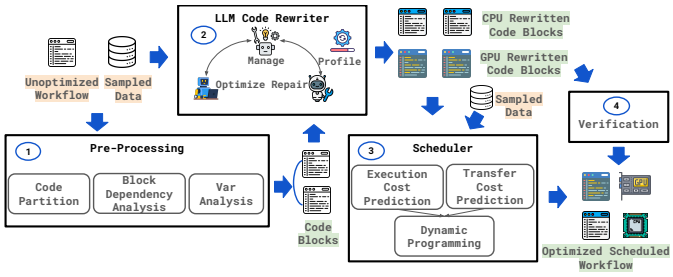
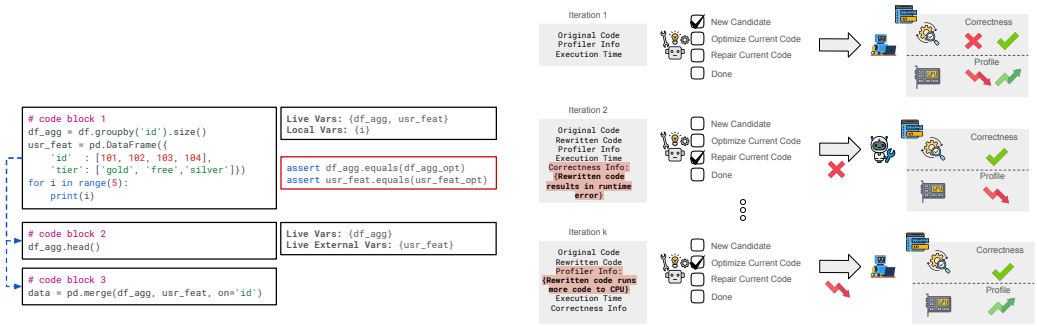


Figure 2: PANDAX's end-to-end workflow.



Left: code block dependency and variable analysis. **Right:** PANDAX's multi-agent rewriting system.

Figure 3: Left: Specification generation and variable analysis. Right: PANDAX's multi-agent system.

4 APPROACH

Fig. 2 summarizes PANDAX's end-to-end workflow. Given a pandas workflow and a small data sample, PANDAX performs pre-processing, backend-specific rewriting, scheduling, and verification to generate an optimized heterogeneous workflow. We describe each stage in detail next.

4.1 PRE-PROCESSING

Before rewriting, PANDAX performs a pre-processing step designed to give the rewriter the necessary context about the workflow's structure and performance. This phase is divided into two parts: dynamic dependency analysis and performance analysis.

First, we partition the input program into a sequence of *code blocks*, which serve as the fundamental units for rewriting and scheduling. For Jupyter notebooks, each cell is naturally treated as a distinct code block. Next, to understand the data flow between code blocks, PANDAX first performs a dynamic dependency analysis. PANDAX executes the original pandas workflow on a provided data sample and traces the lifetime of all data objects (e.g., DF, Series, lists, scalars) across code block boundaries. Inspired by the dependency graph construction Li et al. (2023), our analysis classifies the variables associated with each code block b_i into three categories.

1. **Live Variables.** Created or modified in b_i and then read by some future code block b_j where $j > i$. These variables represent the essential state that must be passed between code blocks.
2. **Local Variables.** Created, used, and go out of scope entirely within b_i. They have no impact on other blocks.
3. **Live External Variables.** Defined in an earlier block b_k ($k < i$) and are not accessed in the current block b_i, but are required by a future block b_j ($j > i$). These variables represent the pass-through state that b_i must preserve.

Fig. 3 illustrates this classification. In code block 1, `df_agg` and `user_feat` are live, while `i` is local. In code block 2, `df_agg` remains live, and `user_feat` is live external since it passes through unused

but is required by code block 3. This structure enables block-local rewriting while preserving global program state.

Following dependency analysis, PANDAX establishes a performance baseline by executing the original workflow on a data sample. For each code block, this involves collecting its total wall-clock time on both the CPU and GPU, as well as detailed operator-level profile information for the GPU execution. This profile reveals which operations fell back to the CPU and the total number of GPU calls (an example is shown in Fig. 1d). It helps identify performance bottlenecks and provides the empirical signal used to guide rewriting and evaluate speedups.

4.2 LLM-BASED CODE REWRITING

With the workflow pre-processed and profiled, PANDAX enters the rewriting phase, where each code block is transformed from its pandas implementation into a performant, backend-specific equivalent.

A prior approach Baziotis et al. (2024) automates such optimization using a pattern-matching-based system. This requires a domain expert to manually define a set of $(pattern, transformation)$ rules. Such systems are brittle to minor code changes and require developing new rule sets for each target backend. PANDAX instead performs rewriting at the code block level within a structured, agent-driven feedback loop. The key insight is that rewriting is not a single-shot translation but an *iterative* process of generation, testing, and refinement.

4.2.1 MULTI-AGENT LLM REWRITING SYSTEM

PANDAX implements the iterative rewriting process via a multi-agent architecture that modularizes code optimization into specialized roles. As shown in Fig. 3, the system comprises three agents: a **Manager**, a **Code Optimization Agent**, and a **Code Repair Agent** (full prompts in **Appendix A**).

Manager Agent. The Manager orchestrates the rewriting process for each code block, maintaining state across iterations and coordinating optimization based on correctness and performance feedback. It controls the rewrite loop by deciding when to explore new rewrite strategies, refine existing candidates, trigger repairs, or terminate once a correct and performant implementation is found.

Code Optimization Agent. The Code Optimization Agent generates backend-targeted, high-performance code for CPU and GPU execution. It uses profiling signals (e.g., runtimes, fallbacks) and backend-specific optimization guidance to propose increasingly efficient implementations using techniques such as vectorization.

Code Repair Agent. When a candidate fails correctness verification, the repair agent analyzes the failure and tries to fix the faulty code.

4.2.2 SPECIFICATION AND CORRECTNESS CHECK

Following each rewrite attempt, PANDAX checks the correctness of the rewrites. A fundamental challenge with these workflows is the lack of formal specifications; the code itself serves as the specification. Checking correctness after rewriting an entire workflow is intractable, as it is difficult to pinpoint where an error might have been introduced. PANDAX addresses this by enforcing *block-local functional equivalence*: each rewritten block b_i' must be a drop-in replacement for the original block b_i at code block boundaries.

Specification. Using dependency analysis from pre-processing, PANDAX defines correctness in terms of state invariants. A rewritten block b_i' is correct relative to b_i if, given the same input sample, it satisfies the following:

1. **Live variable equivalence:** all live variables produced by b_i' are equivalent to those produced by b_i.
2. **External variable preservation:** no live external variables are modified or redefined.

Correctness checking. PANDAX enforces this specification by executing both the original and rewritten code block on the sampled input. It compares primitive values by direct equality, while DFs and Series are checked for equivalence in shape, index, column types, and values. Finally, side-effect operations (e.g., `df.info()`, `df.describe()`) are validated by comparing captured standard output.

As illustrated in Fig. 3, dependency analysis identifies `df_agg` and `usr_feat` as live variables for code block 1. These variables therefore define the correctness assertions for that block. This automated checking mechanism forms the backbone of PANDAX's rewriting loop, ensuring that performance optimizations preserve workflow semantics.

4.2.3 ACCEPTING REWRITE CANDIDATES

Upon correctness verification, PANDAX evaluates performance using a heuristic acceptance model. Because small-sample wall-clock timings are noisy, the model combines observed runtime with operator-level GPU profiling, using the GPU-dispatch ratio r as a proxy for scalable parallelism. A candidate is accepted if it improves performance, rejected if it regresses, and otherwise accepted only when speedup or increased GPU utilization justifies the trade-off. See **Appendix M** for formal logic.

4.3 SCHEDULER

After the rewriting phase, PANDAX consists of a set of locally optimized code blocks, each with a performant implementation for both CPU and GPU. The scheduler's objective is to construct a *global execution plan* that minimizes the end-to-end workflow time by jointly optimizing computation and data movement across heterogeneous backends.

Naive strategies fail in heterogeneous settings. Greedy GPU-first execution incurs frequent CPU fallbacks and repeated data transfers, while per-block best-device assignment ignores cross-block data-movement costs, leading to suboptimal global plans. Both approaches make myopic decisions.

PANDAX instead performs *global scheduling* using a dynamic programming algorithm that minimizes cumulative execution time by jointly considering: 1. execution cost of each code block on both the CPU and GPU, and 2. transfer cost of live data between device memories.

Since all profiling is performed on small samples, PANDAX first learns cost models to extrapolate full-dataset execution and transfer costs, and then applies DP to compute the optimal plan. We assume a single-GPU setting in which live data for any GPU-scheduled code block fits in device memory.

4.3.1 LEARNED EXECUTION COST MODEL

To predict full-dataset execution time, PANDAX builds a temporary, per-block cost model rather than relying on static operator models. Static operator models attempt to pre-learn performance for individual pandas APIs (e.g., `groupby`, `join`, `apply`) across backends, but this approach is intractable due to the combinatorial space induced by operator compositions, parameterizations, data distributions, and arbitrary Python interleaving. Instead, for each rewritten code block, PANDAX executes the CPU and GPU implementations on multiple sub-samples of different sizes and fits a simple linear model: $\text{time}(size) = m \cdot size + c$, where m captures marginal per-row cost and c captures fixed overheads (e.g., kernel launch, initialization). This enables reliable extrapolation from small samples to full datasets and provides accurate cost estimates for scheduling. As we show in **Appendix D**, these simple linear models are sufficient to accurately capture scheduling trade-offs.

4.3.2 TRANSFER COST MODEL

PANDAX models GPU↔CPU transfer costs using offline-trained, data-type–aware linear models. Separate models are learned for common column types (e.g., `int64`, `float64`, `bool`, `datetime64`, `category`, `object/string`), capturing allocation, serialization, and conversion overheads. Transfer cost is modeled as $t_{\text{transfer}} = m \cdot \text{bytes} + n \cdot \text{rows} + c$, where m captures per-byte transfer cost, n models per-element processing overhead (for variable-length types), and c represents fixed per-call latency. These models enable accurate prediction of transfer costs for live data objects moved between devices. Note that PANDAX only transfers required columns when moving data between devices.

4.3.3 DYNAMIC PROGRAMMING ALGORITHM

Using the learned execution and transfer models, PANDAX solves the global scheduling problem in Eq. (1) via dynamic programming. The state of our DP captures the current block index and the memory layout of live data, rather than all intermediate values, which keeps the DP tractable.

The recurrence follows directly from Eq. (1): the total cost of executing the workflow starting from block b_i on device $d \in \{\text{CPU}, \text{GPU}\}$ is the sum of the required transfer cost from block b_{i-1} to b_i, the execution cost of b_i on d, and the optimal cost of executing the remaining blocks. At each step, the scheduler evaluates both device choices and selects the minimum-cost plan.

We use memoization to avoid redundant evaluations and backtracking to yield the optimal heterogeneous execution plan. This formulation enables PANDAX to compute globally optimal device placements that minimize total workflow time, rather than relying on greedy or locally optimal heuristics. Full algorithmic details, state formulations, and an empirical analysis of scheduler scaling and scheduling costs are provided in **Appendices J** and **L**.

4.4 VERIFICATION OF REWRITES

As described in Sec. 4.2.1, PANDAX initially checks correctness using sample-based execution. However, empirical testing cannot guarantee semantic equivalence over all possible inputs. Full formal verification of arbitrary Python and pandas programs remains an open research problem Ding et al. (2023); Wang et al. (2024); Chu et al. (2017); Zhao et al. (2025); Wang et al. (2022) due to complex data types, imperative loops, and massive API surface areas.

To bridge the gap between empirical testing and full formal proof, PANDAX introduces a *bounded formal verification* step before finalizing the workflow. Inspired by recent work on bounded equivalence checking for SQL He et al. (2024), PANDAX performs verification at the code block level.

Symbolic Encoding & Bounded Checking. We encode pandas semantics as symbolic constraints in SMTLIB Barrett et al. (2010) using the CVC5 solver. `Series` and `DataFrame` objects are modeled as collections of symbolic variables. To ensure tractability, complex data types are abstracted into SMT-native representations (e.g., categorical strings to integers, `NaN` to reserved constants, and variable-length strings to bounded symbolic sequences). We encode the semantics of relational operations, filtering, and aggregations as logical constraints, while arbitrary Python UDFs and external libraries are modeled safely as uninterpreted functions.

Following the same specification used during our sample-based testing, we define equivalence strictly over the block's *live variables*. For an original block b_i and its rewrite b_i', we symbolically execute both on an input DataFrame of symbolic size k. **The "bounded" nature of our verification refers to this symbolic row limit k; we prove equivalence for all possible inputs up to k rows.** We then assert the negation of output equality for all live variables ($\neg(O_i = O_i')$). A satisfying model from CVC5 yields a concrete counterexample, whereas unsatisfiability *formally proves* functional equivalence for all possible inputs up to bound k. We provide complete details on our SMT formulation, explicit solver assumptions, verification scaling, and a concrete string-transformation example in **Appendix K**.

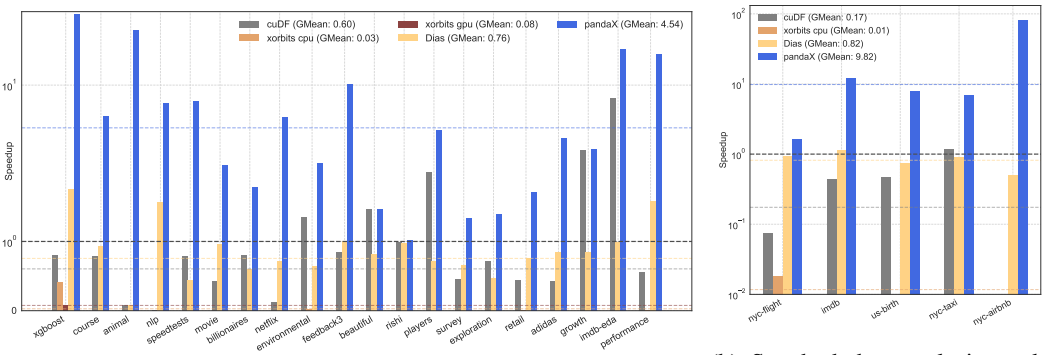
5 EXPERIMENTS

We evaluate PANDAX on three benchmark suites: 20 real-world Kaggle notebooks from Dias Baziotis et al. (2024), 5 standard large-scale data-analysis workflows (NYC Taxi kaggle (c), Flights kaggle (b), Airbnb kaggle (a), U.S. Births kaggle (d), and IMDB Flannery (2016)), and 22 TPC-H decision-support queries implemented in pandas tpch org; xorbits (2022).

5.1 IMPLEMENTATION

We run all our experiments on a dedicated server with Intel Xeon CPUs (Broadwell 2.2GHz) with 8 vCPUs and 52 GB of memory, an NVIDIA Tesla V100 GPU with 16 GB of VRAM, and software frameworks including pandas 2.2.3 and cuDF 25.02. In addition, we evaluate PANDAX on NVIDIA L40S; see **Appendix F**.

We implement PANDAX's multi-agent system using OpenAI's Agent SDK OpenAI. We use the GPT-4o model for the Manager Agent and the o4-mini model for the Code Optimization and Code Repair Agents with temperature 1.0. We use CVC5 Barbosa et al. (2022) for the bounded verification check. Curve fitting and transfer cost models for the scheduler are trained using scikit-learn. We sample 30-60% of the original data for curve fitting, and 50% to estimate transfer costs. **Appendix G** details PANDAX overheads (optimization time and API costs), dataset statistics, and variance.



(a) Dias benchmarks.

(b) Standard data analysis workflows.

Figure 4: Comparison of GPU speedups over pandas. Dotted lines show geomean vs pandas.

5.2 Baselines

We compare PandaX against four baselines: pandas McKinney (2010), which executes the original workflows on the CPU; cuDF cuDF (2018), a GPU-accelerated drop-in DataFrame library; Dias Baziotis et al. (2024), the prior state-of-the-art rule-based rewriter for pandas; and xorbits Lu et al. (2024), a framework for accelerating pandas workloads on multi-core CPUs and GPUs with minimal code changes. We focus on systems that operate directly on original pandas code, excluding Polars, whose API requires manual porting, and Modin, which is less competitive than the more directly comparable rewriting baseline Dias.

5.3 Dias Benchmarks

We first evaluate PandaX on the 20 real-world data science workflows curated by Dias Baziotis et al. (2024). This benchmark suite is ideal for our evaluation as it consists of Kaggle notebooks, representing complex and often inefficient pandas code written by data scientists. Fig. 4a illustrates speedups over pandas for cuDF, Dias, and PandaX. PandaX achieves a geometric mean speedup of **4.5**×, with a peak speedup of **37**×, consistently improving performance across diverse workflows.

Naive GPU offloading fails. The cuDF baseline (greedy GPU-first execution) achieves a geometric mean slowdown of $0.60\times$, degrading performance on 14/20 benchmarks. This reflects common GPU inefficiencies in real workflows, including excessive host-device transfers, unsupported operators, and non-vectorized access patterns. The `nlp` benchmark illustrates this starkly: cuDF times out after 3 hours due to loop-heavy execution, while PandaX completes it in **14s**. Overall, PandaX achieves a **7.4**× geomean speedup over cuDF.

Rule-Based Systems are Brittle. Dias improves only 3/20 workflows, resulting in a geomean slowdown of $0.76\times$[1], demonstrating the fragility of pattern-matching approaches. PandaX achieves a **6**× geomean speedup over Dias. Executing Dias-generated code on GPU yields performance nearly identical to naive cuDF, confirming that most code remains unoptimized.

Drop-in compatibility is insufficient. xorbits executes only 4/20 workflows on CPU and 1/20 on GPU; the rest fail due to unsupported operations or runtime errors. For the few that run, their performance is substantially worse than pandas (geomean slowdowns of $0.03\times$ on CPU and $0.08\times$ on GPU), indicating that semantic preservation without structural rewriting is ineffective for performance.

5.4 Standard Data Analysis Workflows

We evaluated PandaX on five additional publicly available data-analysis workflows on large datasets such as NYC taxi, flights, and Airbnb. As shown in Fig. 4b, PandaX achieves a geometric mean speedup of $9.82\times$ over pandas, with a peak speedup of $82\times$ on `nyc-airbnb`.

[1]Our reported numbers for Dias differ slightly from those in the original paper due to environmental differences; we reran all baselines on our specific hardware and software stack (pandas 2.2.3 vs. their 1.5.1)

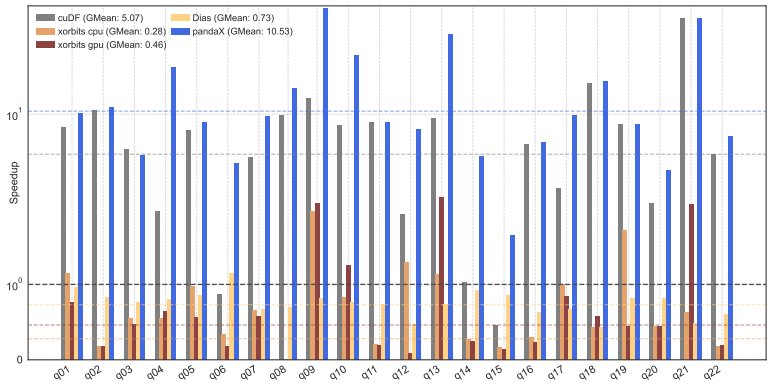


Figure 5: Speedup over pandas for TPC-H (factor 10).

The baselines again perform poorly. cuDF yields a geomean slowdown of $0.17\times$ and times out on `nyc-airbnb`, where PANDAX rewrites a row-wise loop into a vectorized form and completes in 1.1s. Dias improves only one workflow, with a geomean slowdown of $0.82\times$, while xorbits executes only one workflow on CPU and none on GPU.

To further demonstrate that both semantic rewriting and global scheduling are essential for scalable performance gains, we ablate on these two components in **Appendix C**. In addition, we provide qualitative examples for both components in **Appendix H** and **Appendix I**.

5.5 TPC-H QUERIES

We evaluate PANDAX on the standard TPC-H benchmarks implemented in pandas. Rewrites are learned using factor 1 data and evaluated on factor 10 and factor 15 (presented in **Appendix E**) datasets; results for factor 10 are shown in Fig. 5.

PANDAX achieves a geometric mean speedup of **10.5**$\times$ over pandas, outperforming cuDF (**5**$\times$ geomean). Although cuDF is already effective on TPC-H, PANDAX consistently discovers additional optimization opportunities, demonstrating benefits even on structured, analytics-style workloads.

Dias rewrites only one query (`q06`), yielding a geomean slowdown of $0.73\times$, reflecting the brittleness of pattern-matching approaches on complex queries. xorbits executes most queries but results in geomean slowdowns of $0.46\times$ (GPU) and $0.28\times$ (CPU), inheriting cuDF bottlenecks.

6 RELATED WORK

GPU DataFrame Libraries. Systems such as cuDF cuDF (2018) and hipDF hipDF (2025) accelerate pandas execution, but their GPU-first policies perform poorly on heterogeneous workflows due to fallbacks and data-transfer overheads. PANDAX instead rewrites workflows and globally schedules execution across CPU/GPU backends.

Code Rewriting. Prior work includes rule-based rewriting Baziotis et al. (2024), search-based synthesis Bhatia et al. (2023); Cheung et al. (2013); Qiu et al. (2024), and LLM-based code generation Bhatia et al. (2025); Huang et al. (2024); Chen et al. (2021); Taneja et al. (2025). These methods are brittle, constrained, or lack guarantees. PANDAX instead combines LLM-driven rewriting with formal verification, performance modeling, and global scheduling.

Data Science Agents. Recent work Nam et al. (2025b); Li et al. (2024); Nam et al. (2025a) generates data science workflows from natural language; PANDAX complements these systems by optimizing the resulting workflows.

7 CONCLUSION

PANDAX addresses a critical challenge in data science by automating heterogeneous optimization for data science workflows, substantially outperforming prior systems. More broadly, PANDAX takes a first step toward bounded verification of pandas rewrites, connecting LLM-based code generation with formal guarantees for data science pipelines.

REFERENCES

The state of data science 2024: 6 key data science trends. https://blog.jetbrains.com/pycharm/2024/12/the-state-of-data-science/#. Accessed: 2026-05-06.

Pypi stats. https://pypistats.org/packages/pandas. Accessed: 2026-05-06.

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 415–442. Springer, 2022.

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, pp. 14, 2010.

Stefanos Baziotis, Daniel Kang, and Charith Mendis. Dias: Dynamic rewriting of pandas code. *Proc. ACM Manag. Data*, 2(1), March 2024. doi: 10.1145/3639313. URL https://doi.org/10.1145/3639313.

Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In Karim Ali and Guido Salvaneschi (eds.), *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 38:1–38:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.38. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.38.

Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Verified code transpilation with llms. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc. ISBN 9798331314385.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. *SIGPLAN Not.*, 48(6):3–14, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462180. URL https://doi.org/10.1145/2499370.2462180.

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for sql. In *CIDR*, pp. 1–7, 2017.

cuDF. cudf framework. https://rapids.ai/cudf-pandas/, 2018. Accessed: 2026-05-06.

dask. dask framework. https://www.dask.org/, 2015. Accessed: 2026-05-06.

Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. Proving query equivalence using linear integer arithmetic. *Proc. ACM Manag. Data*, 1(4), December 2023. doi: 10.1145/3626768. URL https://doi.org/10.1145/3626768.

Selena Flannery. Probability and movies: Looking at data from IMDB. https://github.com/SeleniumK/data_analysis_and_visualization/blob/fddd9fcd8b4eec6dff43b2fdd7660e316ed142c7/imdb-prob.ipynb, 2016. Jupyter notebook. Accessed: 2026-09-25.

Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. Verieql: Bounded equivalence verification for complex sql queries with integrity constraints. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1071–1099, 2024.

hipDF. hipdf framework. https://github.com/ROCm-DS/hipDF, 2025. Accessed: 2026-05-06.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

kaggle. new-york-city-airbnb-open-data. https://www.kaggle.com/datasets/dgomonov/new-york-city-airbnb-open-data, a. Accessed: 2026-05-06.

kaggle. nyc flight analysis. https://www.kaggle.com/datasets/aephidayatuloh/nyc-flights-2013, b. Accessed: 2026-05-06.

kaggle. nyc-trip. https://www.kaggle.com/datasets/elemento/nyc-yellow-taxi-trip-data, c. Accessed: 2026-05-06.

kaggle. us-birthsdata-analysis. https://www.kaggle.com/code/milanpatra/us-birthsdata-analysis/notebook, d. Accessed: 2026-05-06.

Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. Elasticnotebook: Enabling live migration for computational notebooks. *Proc. VLDB Endow.*, 17(2):119–133, October 2023. ISSN 2150-8097. doi: 10.14778/3626292.3626296. URL https://doi.org/10.14778/3626292.3626296.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, Wanjun Zhong, Wangchunshu Zhou, Wenhao Huang, and Ge Zhang. Autokaggle: A multi-agent framework for autonomous data science competitions, 2024. URL https://arxiv.org/abs/2410.20424.

Weizheng Lu, Kaisheng He, Xuye Qin, Chengjie Li, Zhong Wang, Tao Yuan, Xia Liao, Feng Zhang, Yueguo Chen, and Xiaoyong Du. Xorbits: Automating operator tiling for distributed data science. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 5211–5223, 2024. doi: 10.1109/ICDE60146.2024.00392.

Wes McKinney. Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference*, pp. 56–61, 2010. doi: 10.25080/Majora-92bf1922-00a.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, and Tomas Pfister. Ds-star: Data science agent via iterative planning and verification, 2025a. URL https://arxiv.org/abs/2509.21825.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Arık, and Tomas Pfister. Mle-star: Machine learning engineering agent via search and targeted refinement, 2025b. URL https://arxiv.org/abs/2506.15692.

OpenAI. Openai sdk. https://github.com/openai/openai-agents-python. Accessed: 2026-05-06.

Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(12):2033–2046, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407807. URL https://doi.org/10.14778/3407790.3407807.

polars. polars framework. https://pola.rs/, 2020. Accessed: 2026-05-06.

Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations. In Jonathan Aldrich and Guido Salvaneschi (eds.), *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 32:1–32:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-341-6. doi: 10.4230/LIPIcs.ECOOP.2024.32. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.32.

Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. Llm-vectorizer: Llm-based verified loop vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, pp. 137–149, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712753. doi: 10.1145/3696443.3708929. URL https://doi.org/10.1145/3696443.3708929.

tpch org. Tpc-h. https://www.tpc.org/tpch/. Accessed: 2026-05-06.

Shuxian Wang, Sicheng Pan, and Alvin Cheung. Qed: A powerful query equivalence decider for sql. *Proc. VLDB Endow.*, 17(11):3602–3614, July 2024. ISSN 2150-8097. doi: 10.14778/3681954.3682024. URL https://doi.org/10.14778/3681954.3682024.

Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. Wetune: Automatic discovery and verification of query rewrite rules. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pp. 94–107, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526125. URL https://doi.org/10.1145/3514221.3526125.

xorbits. Tpc-h pandas queries. https://github.com/xorbitsai/xorbits/tree/main/benchmarks/tpch, 2022. Accessed: 2025-10-15.

Fuheng Zhao, Jiayue Chen, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. Llm-sql-solver: Can llms determine sql equivalence?, 2025. URL https://arxiv.org/abs/2312.10321.

Appendix

A PROMPTS

Manager Agent Prompt

You are an expert in code optimization. The user will provide code and context; your goal is to analyze and decide the next step based on the current iteration and remaining iterations. **Options:**

`"1"` `"optimize"`: Further optimizes the rewritten code (only if a rewrite exists).

`"2"` `"repair"`: Fix errors in the rewritten code.

`"3"` `"new"`: Ignore the rewritten code and generate a fresh optimization of the original.

`"4"` `"done"`: No further optimization needed.

General Instructions for Decision Making:

1. Even if the code already uses cuDF or has been sped up, if more efficient APIs or patterns, choose `"optimize"` or `"new"`.
2. If the code's primary purpose is debugging/printing (e.g. `.head()`, `.info()`, `.describe()`, etc.), it cannot be meaningfully optimized.
3. If no rewritten code is present, select `"new"`.
4. If the code is beyond repair, select `"new"`.

Figure 6: Prompt for Manager Agent

GPU Optimizer Agent Prompt

You are an expert in optimizing pandas workflow to execute using cuDF. We are going to show you one cell from the notebook at a time and your task is to rewrite a given code snippet for cuDF. Carefully analyze the provided code and the profiling data to identify inefficiencies. **General Instructions:**

1. The notebook already has the (%load_ext cudf.pandas) extension loaded. We can directly use pandas APIs and don't need to replace them with cuDF APIs.
2. Don't import cuDF in the code and don't manually move data between CPU and GPU as cuDF will handle it automatically.
3. Don't use any other external libraries.
4. If no modifications are made to the original code, return code as 'done'.
5. The CPU operations are listed in the profiling data under the CPU calls column.
6. The GPU operations are listed in the profiling data under the GPU calls column.

Code Optimization Instructions: Try to optimize the code as much as possible to run with cuDF. This can be achieved by:

1. Eliminate loops in favor of vectorized operations.
2. Minimize the data movement between CPU and GPU by replacing CPU operations with GPU operations.
3. If the code is already using GPU, try to optimize it further by using more efficient GPU operations or exploring alternative APIs.
4. Rewritten code should be semantically equivalent to the original code.
5. If the code is complex or has multiple loops, try to optimize it piece by piece as there are multiple iterations for optimization.

Var Dependency Instructions:

1. Active vars are the variables that are modified in the original code. These variables can be rewritten but need to be maintained in the optimized code.
2. Future vars are the variables that are being modified in the next code cells. These variables cannot be reassigned in the optimized code or modified in-place.

CPU Optimizer Agent Prompt

You are an expert in optimizing pandas code for CPU efficiency. Your job is to rewrite code to be faster and more memory-efficient while keeping its behavior exactly the same. **General Instructions:**

1. Don't use any other external libraries.
2. pandas is already imported and can be used with pd.
3. If no modifications are made to the original code, return code as 'done'.

Code Optimization Instructions: Try to optimize the code as much as possible for running with pandas. This can be achieved by:

1. Eliminate loops in favor of vectorized operations.

2. Chain operations to avoid intermediate copies.
3. Avoid recomputation.
4. Rewritten code should be semantically equivalent to the original code.
5. If the code is complex or has multiple loops, try to optimize it piece by piece as there are multiple iterations for optimization.

Var Dependency Instructions:

1. Active vars are the variables that are modified in the original code. These variables can be rewritten but needs to be maintained in the optimized code.
2. Future vars are the variables that are being modified in the next code cells. These variables cannot be reassigned in the optimized code or modified in-place.

Figure 7: Prompt for Optimizer Agents

GPU Repairer Agent Prompt

Your task is to fix the code for errors. We are trying to optimize pandas workflow to execute using cuDF.

General Instructions:

1. The notebook already has the (%load_ext cudf.pandas) extension loaded. We can directly use pandas APIs and don't need to replace them with cuDF APIs.
2. Don't import cuDF in the code and don't manually move data between CPU and GPU as cuDF will handle it automatically.
3. Don't use any other external libraries.
4. The CPU operations are listed in the profiling data under the CPU calls column.
5. The GPU operations are listed in the profiling data under the GPU calls column.

Code Optimization Instructions: Try to optimize the code as much as possible for running with cuDF. This can be achieved by:

1. Eliminate loops in favor of vectorized operations.
2. Minimize the data movement between CPU and GPU by replacing CPU operations with GPU operations.
3. If the code is already using GPU, try to optimize it further by using more efficient GPU operations or exploring alternative APIs.
4. Rewritten code should be semantically equivalent to the original code.
5. Try to repair the part of the code only for which the assertions are failing. Do not make any changes to the code that is not related to the assertions.

CPU Repairer Agent Prompt

You are an expert in optimizing pandas code for CPU efficiency. Your job is to rewrite code to be faster and more memory-efficient while keeping its behavior exactly the same.

General Instructions:

1. Don't use any other external libraries.
2. pandas is already imported and can be used with pd.

Code Optimization Instructions: Try to optimize the code as much as possible for running with pandas. This can be achieved by:

1. Eliminate loops in favor of vectorized operations.
2. Chain operations to avoid intermediate copies.
3. Avoid recomputation.
4. Rewritten code should be semantically equivalent to the original code.
5. Try to repair the part of the code only for which the assertions are failing. Do not make any changes to the code that is not related to the assertions.

Figure 8: Prompt for Repair Agent

B Executive Summary of Additional Evaluations

To supplement the primary evaluation in Sec. 5, we conducted extensive ablations, scalability tests, and cost analyses. Table 1 provides a high-level summary of the key empirical findings detailed in the subsequent sections of this appendix.

Table 1: Executive Summary of Appendix Evaluations and Ablations.

Experiment / Analysis	Key Finding & Metric
Ablation: No Scheduler *(Sec. C.0.1)*	Greedy placement fails to match the DP scheduler. Forcing CPU-only rewrites yields a **1.61**× speedup, and GPU-only yields **2.05**×. The full PANDAX system intelligently schedules across both to achieve **4.5**×.
Ablation: No Rewriter *(Sec. C.0.2)*	Scheduling unoptimized code provides minimal benefits (**1.54**× speedup), proving that LLM-based semantic rewriting is essential for unlocking backend performance.
Hardware Generalization *(Sec. F)*	PANDAX easily generalizes to new hardware. When evaluated on an NVIDIA L40S GPU (requiring only a 1-day retraining of the transfer cost model), PANDAX achieved a **9.29**× geomean speedup (peak **180**×).
Larger Data: TPC-H SF 15 *(Sec. E)*	On a larger scale factor of 15 (where 12/22 pandas queries OOM on CPU), PANDAX maintains dominance, achieving an **8.5**× speedup over pandas, compared to naive cuDF's 6.02×.
Optimization Cost *(Sec. G)*	The offline rewriting phase is highly practical: median optimization time is **232 seconds** per workflow, costing roughly **$50 to $100** in total API credits to run all the experiments.
Run-to-Run Variance *(Sec. G)*	Performance gains are highly stable. Across two separate runs of 10 sampled benchmarks, the geomean speedup varied minimally from **4.17**× to **4.51**×.

C Ablation Studies

We next perform a series of ablation studies to measure the contribution of each of the two components in our system. We also evaluate the accuracy of our learned cost models. We perform all the ablation experiments with the V100 GPU.

C.0.1 Removing the Scheduler.

In this experiment, we disable the DP scheduler. Instead, we use our rewriter to generate two versions of each workflow: one optimized entirely for multi-core CPU execution (`CPU_rewrite`) and one entirely for the GPU (`GPU_rewrite`). We show the speedups over pandas in Fig. 11a. The GPU-only rewrites achieve a geomean speedup of **2.05**×, while the CPU-only rewrites achieve a **1.61**× speedup. However, the full PANDAX system, which combines these rewrites with our DP scheduler, achieves a substantially higher geomean speedup of **4.5**×. Across all benchmarks, PANDAX schedules 35% of code blocks on the CPU. The scheduler further speeds up the workflow. For instance, for the `netflix` benchmark, the GPU-only rewrite is actually slower than the CPU-only version. PANDAX intelligently navigates this trade-off, scheduling different code blocks on different backends, resulting in a **5.5**× speedup and outperforming both monolithic strategies.

C.0.2 Removing the Rewriter.

In this experiment, we disable the multi-agent rewriting system of PANDAX. In this configuration (`only_scheduler`), we apply our DP scheduling algorithm directly to the original, unoptimized code blocks. We use the full data sample to schedule, i.e., we do not rely on the cost models for execution and transfer times. We show the speedups over pandas in Fig. 11b. This `only_scheduler` configuration achieves a geomean speedup of **1.54**×. This shows that scheduling on its own provides benefits, but the full PANDAX system achieves substantially higher speedup (4.54×). This difference highlights that the scheduler is limited by the quality of the code it is given. For example, for the `xgboost` benchmark, the scheduler alone provides almost no benefit, as the original code is inefficient on both backends. After rewriting, PANDAX achieves a **37**× speedup.

This two-part ablation study confirms that both components are essential. PANDAX's rewriter creates high-quality, backend-specific code, and the DP scheduler intelligently combines them into a globally optimal plan.

D ORACLE EXPERIMENTS.

Next, we perform an ablation to quantify the execution-time and data-transfer prediction errors relative to the overall scheduling performance. We evaluate PANDAX under four cost model configurations:

1. All-Predicted. Both execution and transfer cost models use the predicted values (the default PANDAX configuration).
2. Execution-Oracle. Learned execution costs are replaced with ground-truth runtimes, while transfer costs remain predicted.
3. Transfer–Oracle. Transfer costs are replaced with measured transfer times, while execution costs remain predicted.
4. All-Oracle. Both execution and transfer costs are replaced with ground-truth measurements, representing the optimal "oracle" schedule.

We report the average end-to-end execution time error percentage relative to the All-Oracle setting across 25 benchmarks: 2.4% for All-Predicted, 0.28% for Execution-Oracle, and 1.9% for Transfer–Oracle. In **12** out of the 25 benchmarks, PANDAX's predicted schedule exactly matches the oracle schedule, yielding identical block placements and end-to-end runtimes. For the remaining benchmarks, we observed small deviations. Even in cases where the full schedule differs, over **94%** of individual block placements match the oracle's decisions, indicating robustness of our models. We analyzed the benchmark with the largest discrepancy (≈ 4 sec), where overestimated transfer costs led to suboptimal offloading of intermediate results between CPU and GPU.

E TPC-H QUERIES ON SCALE FACTOR 15

When evaluating with TPC-H queries at a scale factor of 15, memory limitations caused 12 of the 22 queries to fail on the CPU. For the 10 queries that completed successfully, PANDAX demonstrated a geometric mean speedup of $8.5\times$ over pandas. In comparison, running the queries with cuDF without any modifications results in a $6.02\times$ geometric mean speedup.

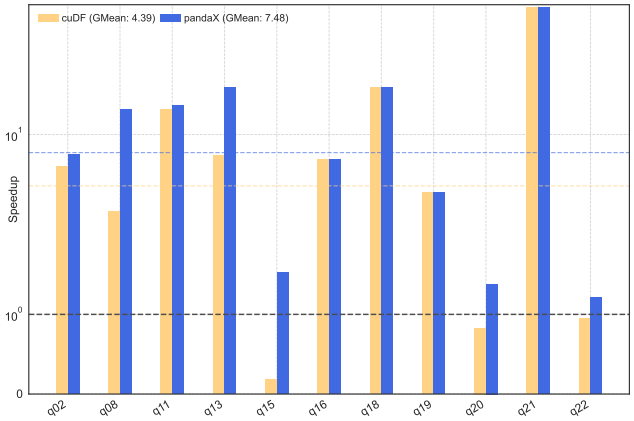


Figure 9: Speedup over pandas for TPC-H (factor 15).

F EVALUATING PANDAX ON ADDITIONAL BACKEND

To test the generalizability of PANDAX, we evaluate it on another backend, the NVIDIA L40S with 48GB of VRAM. The server machine is equipped with an Intel Xeon Gold 6242 CPU operating at a frequency of 2.80GHz, with 64 vCPUs and 504GB of memory. We use the same pandas and cuDF version as the experiments above.

The main backend-specific component of PANDAX is the offline transfer cost model. For this experiment, we retrain the cost models per data type for the L40S GPU using the same approach described in Sec. 4.3, while the rest of the pipeline remains the same. We evaluate PANDAX on the 25 benchmarks (Dias and Data Science Workflows). As shown in Fig. 10, PANDAX achieves an impressive geometric mean speedup of **9.29**$\times$ over pandas, with a peak speedup of **180**$\times$. The speedups observed are again due to the combination of our rewriter and scheduler, similar to those in our previous experiment with the V100 GPU.

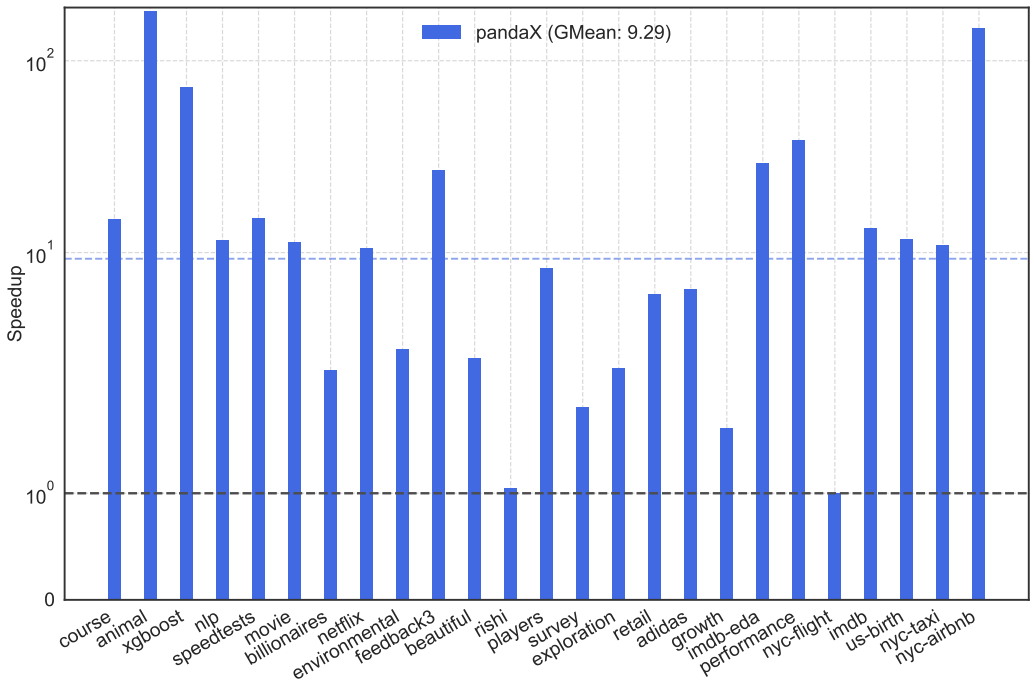
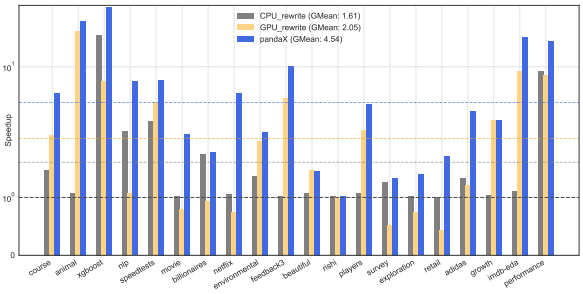
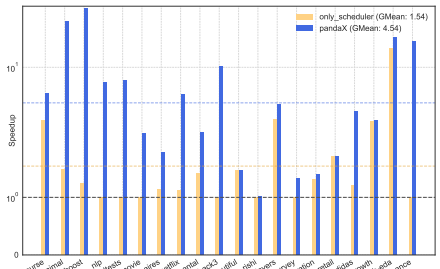


Figure 10: Speedup over pandas on all the 25 notebooks with L40S GPU.

These results show that PANDAX generalizes to new hardware backends. This adaptability stems from PANDAX's modular design; the core optimization steps, i.e., LLM-based rewriter and the DP scheduler, remain unchanged. Extending PANDAX to a new backend only requires training the backend-specific cost models once per data type, using programmatically generated synthetic data. This simple and automated approach allows PANDAX to easily extend to new hardware with minimal engineering effort: adding support for the L40S did not require any code changes to our V100 setup and training the transfer cost models took less than a day.



(a) Speedup over pandas for PANDAX without the scheduler.



(b) Speedup over pandas for PANDAX without the rewriter.

Figure 11: Ablation studies: (a) without scheduler, (b) without rewriter.

G DATASET CHARACTERISTICS, OPTIMIZATION COST, AND VARIANCE

This appendix provides extended details regarding the dataset scale, baseline runtimes, system overhead, and run-to-run variance, supplementing the experimental setup in Section 5.

G.1 DATASET CHARACTERISTICS AND BASELINE RUNTIMES

Our benchmark suite encompasses a highly diverse set of data types (numeric, categorical, and text-heavy) and dataset sizes to ensure robust evaluation. We summarize the dataset sizes and the

unoptimized (base pandas and cuDF) execution times in Table 2. PANDAX consistently demonstrated effectiveness across all data types regardless of scale.

Table 2: Dataset Size and Baseline Run Time Statistics. Note that naive GPU run times (cuDF) often exceed CPU times due to greedy offloading, which triggers expensive operator fallbacks and excessive host-device transfers.

Metric	Dataset Size (MB)	CPU Run Time (s)	GPU Run Time (s)
Average	2089.32	51.00	148.00
Median	2339.74	16.46	23.09
Minimum	38.26	2.50	1.73
Maximum	4909.90	400.00	1726.00

G.2 OPTIMIZATION COST AND RUN-TO-RUN VARIANCE

PANDAX incurs a one-time offline optimization cost during the rewrite stage, after which the optimized code and schedule can be reused across subsequent executions of the workflow. Since code blocks are rewritten independently, this process is naturally parallelizable; under parallel execution, the wall-clock optimization time for a workflow is determined by the slowest block rather than the sum over blocks. Across our benchmarks, the mean end-to-end rewrite time was 237s per workflow, with a median of 232s, minimum of 155s, and maximum of 479s. On average, 56% of this time was spent waiting for LLM API responses rather than performing profiling or correctness checking.

The optimization phase also incurs token and API costs. Across all workflows, mean token usage was 210k tokens per workflow, with a median of 180k, minimum of 42k, and maximum of 500k; these totals include reasoning tokens. The total API cost across all benchmarks was approximately **$50–100**. Since PANDAX targets recurring workflows that are rerun over time, this optimization cost is amortized across many subsequent executions of the optimized workflow.

Amortization on larger workloads. To evaluate the intended regime of larger or recurring workloads, we scaled the Supermarket and Student Performance benchmarks while keeping PANDAX's one-time optimization overhead essentially unchanged. Table 3 reports execution times and optimization overheads, including remote LLM waiting time. On these scaled workloads, PANDAX achieves execution-only speedups of **72.8**× and **1,354.7**×, respectively, while Dias is slower than the original pandas execution on both. Even after including the complete one-time optimization overhead, the first execution is **4.63**× faster than pandas for Supermarket and **6.87**× faster for Student Performance. Thus, both workloads recover their full optimization cost within the first execution. These results suggest that PANDAX's setup cost is primarily program-dependent, while its benefit increases substantially with workload scale.

Table 3: Amortization on scaled workloads. All times are in seconds. First-run total includes one-time optimization overhead and one execution of the optimized workflow. Speedups in the text are calculated from the displayed runtimes.

Benchmark	pandas execution	PANDAX execution	Optimization overhead	First-run total
Supermarket	1,899.4	26.1	384	410.1
Student Performance	1,761.1	1.30	255	256.3

To estimate run-to-run variability, we reran a subset of 10 benchmarks twice. Due to budget constraints, we did not repeat the full benchmark suite. The geomean speedup over pandas was **4.17**× on the originally reported runs and **4.51**× on the second runs, indicating that the aggregate end-to-end performance gains are reasonably stable across repeated executions.

H QUALITATIVE REWRITING EXAMPLES

H.1 CPU REWRITE EXAMPLES

Table 4 presents representative examples of code optimized for CPU execution by our multi-agent LLM rewriting system. The rewritten code delivers significant performance gains, primarily by

Example 1: Vectorization + Grouping (nyc-taxi)

```python
# Original (18.6s)
nei_x_1 = []
nei_y_1 = []
colors_1 = []
for nei_1 in
df['neighbourhood'].value_counts().index:
    nei_x_1.append(nei_1)
    nei_y_1.append(
        np.mean(df[df['neighbourhood']==nei_1]
        ['price'].values)
    )
for neigh_1 in nei_x_1:
    group = df[df['neighbourhood']==neigh_1]['
    neighbourhood_group'].values[0]
    colors_1.append({
        'Manhattan': 'red','Brooklyn': 'blue',
        'Queens': 'green','Bronx': 'orange',
        'Staten Island': 'purple'}[group])
```

```python
# Rewritten (0.0615s)
counts = df['neighbourhood'].value_counts()
grp = (
    df.groupby('neighbourhood')
      .agg(price_mean=('price', 'mean'),
           neighbourhood_group=('neighbourhood_
            group', 'first'))
      .reindex(counts.index)
)
nei_x_1 = grp.index.tolist()
nei_y_1 = grp['price_mean'].tolist()
_color_map = {
    'Manhattan': 'red', 'Brooklyn': 'blue',
    'Queens': 'green', 'Bronx': 'orange',
    'Staten Island': 'purple'
}
colors_1 = grp['neighbourhood_group'].map(_col
or_map).tolist()
```

Example 2: Favor native operations (performance)

```python
# Original (10.6s)
df['OverAll_PassStatus'] = df.apply(
    lambda x: 'F' if x['Math_PassStatus'] == 'F'
    or
                  x['Reading_PassStatus'] ==
                  'F' or
                  x['Writing_PassStatus'] ==
                  'F' else 'P', axis=1)
df['OverAll_PassStatus'].value_counts()
```

```python
# Rewritten (0.307s)
mask = df[['Math_PassStatus',
           'Reading_PassStatus',
          'Writing_PassStatus']].eq('F').any(a
           xis=1)
df['OverAll_PassStatus'] = mask.map({True: 'F',
False: 'P'})
df['OverAll_PassStatus'].value_counts()
```

Example 3: Avoid duplicate computation (nyc-airbnb)

```python
# Original (0.00396s)
x_2 = df['calculated_host_listings_count']\
    .value_counts().sort_index().index.astype(
    str)
y_2 = df['calculated_host_listings_count']\
    .value_counts().sort_index().values
```

```python
# Rewritten (0.00204s)
tmp = df['calculated_host_listings_count'].val
ue_counts()\
    .sort_index()
x_2 = tmp.index.astype(str)
y_2 = tmp.values
```

Example 4: RegEx String cleaning (adidas)

```python
# Original (0.640s)
df['original_price'] =
df['original_price'].str.split('$')
df['original_price'] =
df['original_price'].str[1]
```

```python
# Rewritten (0.491s)
df['original_price'] =
df['original_price'].str.extract(r'\$([^$]*)',
expand=False)
```

Table 4: Illustrative examples of CPU code rewrites that improve efficiency. Each rewrite eliminates common inefficiencies such as Python loops, repeated computation, or unoptimized lambda functions.

Example 1: Vectorization and branch elimination (nyc-airbnb)

```python
# Original (times out)
df["price_group"] = 0
for idx,pri in enumerate(df['price']):
    if df.loc[idx, 'price'] < 50:
        df.loc[idx, 'price_group'] = 0
    elif 50 <= df.loc[idx, 'price'] < 250:
        df.loc[idx, 'price_group'] = 1
    elif 250 <= df.loc[idx, 'price'] < 500:
        df.loc[idx, 'price_group'] = 2
    elif 500 <= df.loc[idx, 'price'] < 750:
        df.loc[idx, 'price_group'] = 3
    elif 750 <= df.loc[idx, 'price'] < 1000:
        df.loc[idx, 'price_group'] = 4
    else:
        df.loc[idx, 'price_group'] = 5
```

```python
# Rewritten (0.18s)
# Define the bin edges (right=False to match the
original < and <= logic)
bins = [0, 50, 250, 500, 750, 1000,
float("inf")]

# Initialize all groups to 0 (as in the original
loop)
df["price_group"] = 0

# Vectorize the bucketing for the first 100
entries only (to match the original slicing)
df["price_group"] = pd.cut(
    df["price"],
    bins=bins,
    labels=False,
    right=False
).astype(df["price_group"].dtype)
```

Example 2: Use `groupby` in place of `pivot_table` (us-birth)

```python
# Original (4.5s)
birth_data.pivot_table('births',
index='decade', columns='gender',
aggfunc='mean')
```

```python
# Rewritten (0.027)
birth_data.groupby(['decade',
'gender'])['births'].mean().unstack('gender')
```

Example 3: Prefer non-in-place operations (performance)

```python
# Original (3.6s)
course.sort_values("course_rating_mean",
ascending=False, inplace=True)
```

```python
# Rewritten (0.104s)
course =
course.sort_values("course_rating_mean",
ascending=False)
```

Example 4: Eliminate `for` loops (players)

```python
# Original (0.396s)
# get ball snap indicies
idxs = data.loc[data['event']=='ball_snap',
'frameId'].index.values

# to get 500ms of movement after snap, get 5
rows (each row is 100ms of info)
x = [(idxs+x).tolist() for x in range(0,6)]
idxs = [item for sublist in x for item in
sublist] #the output x is a list of lists, so
this is just to flatten the list

# filter for snap + 500ms of data only using our
selected indicies
df = data.reindex(idxs)
```

```python
# Rewritten (0.102s)
# vectorized approach using groupby and cumcount
snap_group =
data['event'].eq('ball_snap').cumsum()
offset = data.groupby(snap_group).cumcount()
# keep rows from each snap (offset 0) up to 5
rows after (offset <= 5)
df = data[(snap_group > 0) & (offset <= 5)]
```

Table 5: Illustrative examples of GPU code rewrites that improve efficiency. Each rewrite eliminates common inefficiencies such as branches, for loops, and in-place operations.

eliminating Python loops, leveraging vectorized operations, chaining transformations, and replacing slower constructs with efficient native primitives. All reported timings are measured within the original notebook environments.

H.2 GPU REWRITE EXAMPLES

Table 5 shows examples of PANDAX-written GPU-optimized code. Common strategies for accelerating pandas code on GPUs include vectorization, eliminating for-loops and branches, avoiding in-place operations, and steering clear of inefficient or unsupported GPU operations. All reported timings are measured within the original notebook environments.

I DETAILED END-TO-END EXAMPLE

To provide a qualitative understanding of how PANDAX operates end to end, we examine the `nlp` benchmark in the Dias suite. In that benchmark, an early code block uses `apply` to compute the length of each tokenized string from a text column:

```
train["len"] = train["text"].apply(lambda x: len(x.split()))
```

This is inefficient as it involves row-wise iteration over the `text` column, which prevents GPU vectorization. Our agent-based rewriter identifies this anti-pattern and replaces it with a fully vectorized variant that leverages GPU-native string operations:

```
train["len"] = train["text"].str.split().list.len()
```

This transformation is substantial, reducing the code block's GPU execution time from 600ms to just **20ms**. Consequently, the scheduler assigns this optimized code block to the GPU.

A subsequent code block in the same workflow presents a different challenge: a dependency on an external library, scikit-learn's `CountVectorizer`, which operates exclusively on CPU-based data structures.

```
for dt in tqdm(train['type'].unique()):
    df = train.query(f"type == '{dt}'")
    texts = df['text'].tolist()
    vec = CountVectorizer(...).fit(texts)
```

A naive "GPU-first" strategy would attempt to execute this loop on the GPU. While our rewriter can optimize parts of the code block, the dependency on the external library forces numerous data transfers from GPU to CPU memory. Our DP scheduler, with its global view of computation and data transfer costs, correctly identifies this pitfall. Consequently, the scheduler makes the globally optimal decision to schedule this entire code block on the CPU, even though parts of it could be accelerated. This scheduling choice reduces the end-to-end runtime for this code block from 19s to just **7s** including data transfers.

J COST-MODEL ACCURACY AND SCHEDULER SCALING

This section provides additional evidence supporting PANDAX's scheduler. Since the scheduler relies on learned execution and transfer cost models, two questions are central: (1) whether these models are accurate enough to support device-placement decisions, and (2) whether the dynamic-programming (DP) scheduler remains practical as workflow size grows.

Cost-model accuracy. PANDAX uses learned cost models to estimate both execution and transfer costs from small samples. Our transfer-cost model achieved R^2 values between approximately 0.85 and 0.95 in offline evaluation, indicating that simple linear models are sufficiently accurate to capture the dominant factors governing data movement cost. While these models are not intended to predict every runtime constant exactly, they are accurate enough to preserve the relative ordering between alternative device placements, which is the quantity most relevant for scheduling. Furthermore, while execution costs are also modeled linearly, we acknowledge that operations such as join, merge, or sort_values can exhibit super-linear scaling. However, across our evaluated benchmarks, we observed that linear extrapolation from small samples reliably preserved the CPU vs. GPU performance gap, ensuring the DP scheduler consistently selected the correct backend.*

Impact on scheduling quality. To evaluate whether model accuracy is sufficient for scheduling, we compared the plan selected by PANDAX against an oracle plan constructed retrospectively from measured full-data costs. The schedule chosen by PANDAX differed from the oracle by only 2.4% on average in end-to-end runtime, and more than 94% of per-block placements matched the oracle. These results indicate that the learned models are accurate enough to support high-quality global scheduling decisions even when they are fit from small samples.

Metric	Value
Transfer-model R^2	0.85–0.95
Mean end-to-end oracle gap	2.4%
Block placements matching oracle	>94%
DP time complexity	$O(n \cdot 2^L \cdot L)$
DP space complexity	$O(n \cdot 2^L)$
Largest workflows handled	>100 code cells

Table 6: Existing evidence supporting PANDAX's scheduler. The oracle gap compares the predicted schedule against a retrospective oracle schedule computed from measured full-data costs.

DP scheduler complexity. Let n denote the number of code blocks in a workflow and let L denote the maximum number of live variables that cross any block boundary. The DP state is indexed by the current block and the placement of live data, yielding worst-case time complexity

$$O(n \cdot 2^L \cdot L)$$

and space complexity

$$O(n \cdot 2^L).$$

This bound is conservative: in practice, notebook-style workflows tend to have modest live frontiers because many temporaries are local to a short span of consecutive cells.

Practical scaling. In our benchmarks, the scheduler was not a bottleneck, even for workflows with more than 100 code cells. It explored a median of **85** states per workflow, and 24/25 workflows explored fewer than 1000 states. Scheduler runtime was correspondingly small: the median runtime was **5.3ms**, and 24/25 workflows completed in under 50ms. We observed one outlier that explored 11,791 states and still completed in 47.3s. If future workflows induce substantially larger live sets, the same formulation can be relaxed naturally to a beam-search variant with beam width k, reducing complexity to $O(n \cdot k \cdot L)$ time and $O(n \cdot k)$ space while providing a tunable efficiency–optimality tradeoff. This reflects PANDAX's design: rewriting dominates optimization cost due to repeated LLM calls, correctness checking, and profiling, while scheduling adds only a lightweight combinatorial pass.

Discussion. Overall, these results suggest that PANDAX does not require perfectly calibrated cost models to be effective. The learned models are sufficiently accurate to preserve the relative ranking of CPU and GPU alternatives, and the resulting schedules closely track those chosen by an oracle with access to full-data measurements. At the same time, the DP scheduler remains practical on the workflows we study, because the number of simultaneously live variables is typically much smaller than the total number of code blocks.

K FORMAL VERIFICATION DETAILS

While large language models are highly capable of generating performant code, they are prone to subtle semantic hallucinations. PANDAX mitigates this by coupling the LLM rewriter with a bounded SMT equivalence checker. In this section, we provide deeper technical details on our solver formulation, its practical limitations, and its empirical scaling.

K.1 THE CHALLENGE OF FULL FORMAL VERIFICATION

A full formal proof of correctness for arbitrary PANDAX rewrites is a significant challenge. This difficulty arises from the need to formally model the extensive pandas API, its seamless interaction with arbitrary Python code and external libraries, and the high computational cost of reasoning over complex data types such as strings. Furthermore, verifying imperative constructs (such as Python `for` loops) requires advanced techniques like loop-invariant synthesis.

Because pandas workflows interleave relational operations with Python control flow, full formal verification is fundamentally undecidable. In fact, even in the strictly restricted domain of pure relational algebra, query equivalence remains an open problem that is actively worked on in the formal methods research community Ding et al. (2023); Wang et al. (2024); Chu et al. (2017); Zhao et al. (2025); Wang et al. (2022). To make the problem tractable and circumvent undecidability, PANDAX relies on a bounded verification setting—specifically, bounding the row size of the symbolic input DataFrame—alongside a specific set of formalized assumptions.

K.2 SMT Encoding and Supported API

We utilize CVC5 as our underlying SMT solver, encoding pandas operations directly via its Python API. Our verifier does not claim to support the entirety of the massive pandas API; rather, we target the practically grounded subset of operators that empirically arise in our data science benchmarks. Currently, PandaX supports bounded verification for:

- **Filtering and selection:** `loc`, `isin`, `notna`, `dropna`, and boolean masking.
- **Element-wise transformations:** `map`, `replace`, `fillna`, `where`, `assign`.
- **Aggregations:** `groupby`, `nunique`, `unique`.
- **String operations:** `str.split`, `str.partition`, `len`.
- **Relational algebra:** Merges, joins, and standard arithmetic/comparison operators.

We do not claim complete formal semantics for all pandas operators. However, our verifier covers practically all common operations including filtering and selection (`loc`, `isin`, `notna`, `dropna`, Boolean masking), element-wise transformations (`map`, `replace`, `fillna`, `where`, `assign`), aggregation (`groupby`, `nunique`, `unique`), string operations (`str.split`, `list.len`, `str.partition`), joins and merges, and standard arithmetic and comparison operators. Across the 25 workflows used in our evaluation, 80% of accepted rewrites were successfully discharged by the SMT verifier at the configured bound; rewrites that timed out or used unsupported constructs fell back to differential testing. The remaining rewrites failed to verify as they contain calls to external libraries (e.g., `sklearn`) that we currently don't support.

This supported subset provides strong functional coverage for real-world code blocks. Because the bulk of practical data wrangling consists of composing these foundational relational, element-wise, and string primitives, PandaX is able to formally verify a highly diverse range of workflows.

K.3 Formal Assumptions and Limitations

To make equivalence checking tractable, our verification engine operates under a specific set of formalized assumptions:

1. **Uniform Column Types:** We assume that `DataFrame` columns do not contain mixed types. Our current SMT encoding maps each symbolic variable to a fixed sort (e.g., `Int` or `String`). Strings are modeled as finite symbolic sequences over an integer alphabet with bounded length, allowing string operations to be encoded within SMT. While mixed types could in principle be modeled using union types, doing so would substantially increase solver complexity. We additionally assume that Python builtins used by the verified code (e.g., `len`, `sum`, and standard string/list operations) are not overwritten. Our symbolic encoding assigns these builtins their standard semantics.
2. **External Libraries and UDFs:** Arbitrary Python user-defined functions (UDFs) passed to `apply()`, as well as calls to external libraries (like NumPy or scikit-learn), are modeled as *uninterpreted functions*. Because their internal semantics are opaque to the verifier, the solver simply assumes that given the exact same input, an uninterpreted function yields the exact same output.
3. **Compile-Time Schema Knowledge:** Verification does not require concrete runtime data values, but it *does* require accurate schema and type information for the live variables. PandaX infers this schema from the small data sample provided during the pre-processing phase.

K.4 Detailed Walkthrough Example: String Operations in SMT

To demonstrate how PandaX proves equivalence for complex data types without relying on concrete execution, consider a rewrite from the `nlp` benchmark that optimizes a row-wise string length calculation into a vectorized GPU-friendly operation:

- **Original (O):** `train["text"].apply(lambda x: len(x.split()))`
- **Rewritten (R):** `train["text"].str.split().list.len()`

A naive formal verification approach would attempt to explicitly model the creation of lists of strings in memory, which quickly leads to state-space explosion in SMT solvers. Instead, PandaX uses a highly optimized symbolic abstraction.

Step 1: Symbolic String Representation

The verifier abstracts the input column `train["text"]` as a bounded sequence of symbolic integers

(representing characters). Let a single string in this column be represented as a sequence $S = [c_1, c_2, \ldots, c_n]$, where $n \leq k$ (the verification bound). The space character (the default delimiter for `split`) is modeled as a reserved constant integer d.

Step 2: Modeling the Original Python AST

For the original code, PANDAX parses the Python AST of the lambda function. It understands that `x.split()` partitions S at every instance of d, and `len(...)` counts the resulting partitions. Instead of instantiating the partitions, PANDAX's semantic model reduces this directly to a counting problem over the character sequence. A word exists at the start of the string if $c_1 \neq d$. A subsequent word exists at index i if the previous character was a delimiter ($c_{i-1} = d$) and the current character is not ($c_i \neq d$). Therefore, the original program O evaluates to the symbolic constraint:

$$O(S) = \mathbf{1}[c_1 \neq d] + \sum_{i=2}^{n} \mathbf{1}[c_{i-1} = d \wedge c_i \neq d] \tag{2}$$

Step 3: Modeling the Pandas API

For the rewritten code, PANDAX maps the chained pandas methods `.str.split().list.len()` to their corresponding internal semantic models. Because PANDAX encodes the native pandas string length operator using the exact same underlying logic (counting word boundaries rather than materializing lists), the rewritten program R evaluates to the identical symbolic constraint:

$$R(S) = \mathbf{1}[c_1 \neq d] + \sum_{i=2}^{n} \mathbf{1}[c_{i-1} = d \wedge c_i \neq d] \tag{3}$$

Step 4: The SMT Proof

To prove equivalence, PANDAX queries CVC5 to find if there exists *any* string sequence S up to length n where the original and rewritten programs produce different results. The solver is fed the negation of equivalence:

$$\exists S \text{ s.t. } O(S) \neq R(S) \tag{4}$$

Because both $O(S)$ and $R(S)$ deterministically reduce to the exact same mathematical formula inside the solver, the constraint $O(S) \neq R(S)$ simplifies to `False`. CVC5 rapidly returns `UNSAT` (unsatisfiable). By returning `UNSAT`, the solver *formally proves* that the two code blocks are semantically equivalent for all possible input strings up to the bound, guaranteeing that the LLM's rewrite is safe to deploy.

K.5 VERIFICATION PERFORMANCE AND SCALING

We verify equivalence at increasing row bounds k (where k is the number of rows in the symbolic input DataFrame). Checking bounded equivalence up to $k = 1000$ guarantees that no edge case bugs exist for any input configuration up to 1,000 rows.

Table 7 reports the average and median verification times, as well as the timeout rate (using a strict 60-second timeout limit), across all code blocks generated in our evaluation.

Table 7: Bounded Verification Times and Timeout Rates using CVC5 (Timeout = 60s).

Bound (k)	**Avg. Time (s)**	**Median Time (s)**	**Timeout Rate (%)**
10	1.473	0.176	0.0
50	1.638	0.313	20.8
100	9.101	0.273	12.5
1000	1.777	1.641	45.8

At $k = 10$, all blocks verify successfully with zero timeouts and a median time of 0.18 seconds. Crucially, the median verification times remain extremely low (< 2 seconds) even at bounds of $k = 100$ and $k = 1000$, demonstrating that the majority of pandas operations can be verified highly efficiently. The timeout rates at higher bounds are primarily driven by highly complex relational joins and aggregations that cause state-space explosion within the SMT solver. When a timeout occurs, PANDAX safely falls back to sample-based execution testing to evaluate the rewrite's correctness.

K.6 EXTENDING COVERAGE.

Extending verifier coverage is largely incremental engineering rather than a redesign of the framework. Each additional pandas operator requires adding a symbolic semantic model, typically on the order of 20–30 lines of code. Simple element-wise transforms and filters are straightforward to support, while operators with dynamically sized outputs, such as `explode` or `melt`, require more work because their output size depends on data values in ways that are less natural for SMT. Arbitrary Python UDFs passed to `apply` are modeled as uninterpreted functions. In our benchmarks, however, we did not observe such UDFs in the accepted rewrites.

L Dynamic Programming Algorithm

```python
def get_schedule_and_cost(
    cpu_times: list[float],
    gpu_times: list[float],
    input_cols: list[set[str]],
    output_cols: list[set[str]],
    transfer_costs: list[dict[str, dict[tuple[str, str], float]]]
) -> tuple[list[str], float]:
    """
    Compute an optimal per-cell schedule (CPU vs GPU) to minimize total exec+transfer cost.

    Arguments:
      cpu_times:    List of length N. cpu_times[i] is the time to run cell i on the CPU.
      gpu_times:    List of length N. gpu_times[i] is the time to run cell i on the GPU.
      input_cols:   List of length N. input_cols[i] is the set of column names read by cell i.
      output_cols:  List of length N. output_cols[i] is the set of column names written by cell i.
      transfer_costs:
           List of length N. For each i, transfer_costs[i][col][(src, dst)]
           gives the time to move column col from device src to dst immediately before cell i.
           Here src/dst is one of {"cpu","gpu"}.

    Returns:
      schedule:   List of length N, each entry "cpu" or "gpu" for where to run cell i.
      total_cost: Total predicted cost (sum of exec times + transfer times).
    """

    N = len(cpu_times)

    # 1) Map each column to its last-read cell index for dropping unused columns
    last_use = build_last_use(input_cols)

    # Storage for back-pointers.
    parent: dict[tuple[int, frozenset], str] = {}

    # 2) DP with memoization: dp(i, loc_map) = min cost from cell i..N-1
    @lru_cache(maxsize=None)
    def dp(i: int, loc_items: frozenset[tuple[str, str]]) -> float:
        # loc_items encodes a map: col → device where it currently lives
        if i == N:
            return 0.0

        # Reconstruct dict for easier updates.
        loc = dict(loc_items)
        best = float('inf')

        # Try both devices for this cell.
        for dev in ("cpu", "gpu"):
            exec_time = cpu_times[i] if dev == "cpu" else gpu_times[i]
            xfer = 0.0
            new_loc = loc.copy()

            # 2a) Transfer any inputs not already on 'dev'
            for col in input_cols[i]:
                src = new_loc.get(col, dev)
                if src != dev:
                    xfer += transfer_costs[i][col][(src, dev)]
                    new_loc[col] = dev

            # 2b) Mark outputs as now residing on 'dev'
            for col in output_cols[i]:
                new_loc[col] = dev

            # 2c) Remove columns no longer needed
            for col in list(new_loc):
                if last_use.get(col, -1) <= i:
                    del new_loc[col]

            # 2d) Recurse
            cost_rest = dp(i + 1, frozenset(new_loc.items()))
            total = exec_time + xfer + cost_rest
            if total < best:
                best = total
                choice = dev

        # Record choice for reconstruction.
        parent[(i, loc_items)] = choice
        return best

    # 3) Run DP from cell 0.
    total_cost = dp(0, frozenset())

    # 4) Reconstruct schedule.
```

```
    schedule = []
    loc_state = frozenset()
    for i in range(N):
        dev = parent[(i, loc_state)]
        schedule.append(dev)

        loc = dict(loc_state)
        for col in input_cols[i]:
            src = loc.get(col, dev)
            if src != dev:
                loc[col] = dev
        for col in output_cols[i]:
            loc[col] = dev
        for col in list(loc):
            if last_use.get(col, -1) <= i:
                del loc[col]
        loc_state = frozenset(loc.items())

    return schedule, total_cost
```

Figure 12: DP-based CPU/GPU scheduler.

State Representation. The DP state is defined as (i, σ), where:

- i is the index of the current computation block $codeblock_i$,
- σ is the memory layout mapping each live column to its current device: $\sigma : (\text{DataFrame}, \text{column}) \to \{\text{CPU}, \text{GPU}\}$.

Recurrence. Let $C(i, \sigma)$ denote the minimum cost to execute blocks $codeblock_i, \ldots, codeblock_n$ given layout σ before executing $codeblock_i$. Then:

$$C(i, \sigma) = \min_{d \in \{\text{CPU},\text{GPU}\}} \Big(T(\sigma, d, codeblock_i) + R_d(codeblock_i) + C(i+1, \sigma') \Big)$$

Transfer Cost. $T(\sigma, d, codeblock_i)$ is computed by iterating over all live input columns of $codeblock_i$ and summing predicted transfer costs for columns whose current device in σ differs from d, using the learned transfer models.

Layout Update. After executing $codeblock_i$ on device d, the new layout σ' is obtained by:

- setting all output columns of $codeblock_i$ to device d,
- updating input columns that were transferred to device d,
- preserving locations of unaffected columns.

Base Case. $C(n+1, \sigma) = 0$ for all σ.

Initialization. The initial layout σ_1 places all input columns on their initial device (typically GPU).

M HEURISTICS FOR ACCEPTING REWRITE

```python
def should_accept(orig, opt, t0, t1, alpha=1, beta=1):
    # GPU-dispatch ratios
    r0 = orig.gpu_calls/(orig.gpu_calls+orig.cpu_calls)
    r1 = opt.gpu_calls/(opt.gpu_calls+opt.cpu_calls)
    # Dominant cases
    if r1 >= r0 and t1 <= t0:     return True
    if r1 <  r0 and t1 >  t0:     return False

    if r1 < r0:  # slower GPU-ratio but faster overall
        return (t0/t1) >= 1 + alpha*(r0 - r1)
    else:        # improved GPU-ratio but slower overall
        return (t1/t0) <= 1 + beta*(r1 - r0)
```

Figure 13: Heuristic used to accept a rewrite candidate. In all experiments, we use fixed values $\alpha = \beta = 1.0$ and do not tune them per benchmark.

The hyperparameters α and β trade off wall-clock speedup against GPU-dispatch ratio (r). In our implementation, both are fixed to 1.0 for all experiments and are not benchmark-specific tuning parameters. The heuristic is intended as a practical tie-breaking rule for noisy small-sample profiling, not as a theoretically derived objective: runtime is the primary signal, while r acts as a proxy for whether a rewrite is likely to scale better on larger inputs. It is only used in the ambiguous cases where runtime and GPU-dispatch ratio disagree; candidates that are strictly better on both metrics are accepted directly, while candidates that are worse on both are rejected directly. *With $\alpha = \beta = 1.0$, the rule is symmetric: a drop in GPU-dispatch ratio must be offset by a proportional speedup, and conversely a modest slowdown is tolerated only when accompanied by a proportional increase in GPU-dispatch ratio.*
It handles four distinct cases:

1. **Improvement.** Faster and more GPU-parallel $\Rightarrow$ accept.
2. **Regression.** Slower and less GPU-parallel $\Rightarrow$ reject.
3. **Speedup vs. Ratio Drop.** Accept if the speedup compensates for the reduction in GPU-dispatch ratio, e.g., a rewrite lowers r slightly but substantially reduces runtime.
4. **Ratio Gain vs. Slowdown.** Accept if the slowdown is small relative to the increase in GPU-dispatch ratio, e.g., a rewrite introduces a GPU operation whose fixed overhead is significant on small inputs but is likely to be amortized on larger data.

N USE OF LARGE LANGUAGE MODELS

During manuscript preparation, we used large language models to assist with proofreading, correcting grammatical and spelling errors, and making minor edits to improve clarity and readability.